Space Optimization of Counting Sort

Aishwarya Kaul
Department of Computer
Science and Engineering
Bharati Vidyapeeth's
College of Engineering
Guru Gobind Singh Indraprastha University
New Delhi, India

ABSTRACT

Optimization of sorting algorithms is an ongoing research and delivers faster and less space consuming algorithms. The Counting sort algorithm is an integer sorting algorithm and is a very simple and effective way to sort numbers based on their key value. It uses three arrays for computation but in a large input set it can consume a significant amount of memory. This paper puts forward a method to reduce the amount of space required to perform the computation. It reduces the number of arrays or memory required for computation by using just two arrays instead of three, i.e. the input and the count array, removing the need of the third output array.

General Terms

Algorithm; Space Complexity; Optimization

Keywords

algorithms; counting sort; design; optimization; performance; sorting

1. INTRODUCTION

A sorting algorithm is any technique to that arranges a set of inputs in an orderly fashion. The basic aim of all sorting algorithms is of course, to sort the input set in the required manner but, efficiency is also an important factor. In today's world where all aspects of technology are being optimized to give faster results and occupy less memory to entertain more data, it is very important to optimize the algorithms used, that is to sort the input in the fastest and most efficient possible way. The speed of computation or amount of space used by an algorithm during a single computation is referred to by the word "Complexity" or time complexity and space complexity respectively. The complexity of an algorithm is denoted by asymptotic notations such as the big oh O (), theta ϴ () and omega Ω () notation.

Definitions:

1. f(n) = O(g(n)), if there exist constant $c > 0$ and N such that $f(n) <= cg(n)$ for all $n >= N$
2. f(n) = Ω(g(n)), if there exist constant c and N such that $f(n) >= cg(n)$ for all $n >= N$
3. f(n) = ϴ(g(n)), if f(n) = O(g(n)) and g(n) = O(f(n)) [1]

Big oh gives the worst case complexity or the upper bound on the complexity of an algorithm. Omega gives the best case or lower bound whereas theta denotes the average performance of an algorithm or the average case complexity. Nowadays, apart from performance analysis, an energy consumption parameter is also being introduced. It shows that the better performing algorithms also tend to consume more energy. Paper [2] shows more energy savings may be achieved with proper selection of task granularity.

Counting Sort is a non-comparative integer sorting algorithm that means it is used for sorting an integer input set. It can be extended to sorting negative numbers as well. Integer sort is a class of sorting algorithms wherein, the largest element is polynomially bounded in the number of elements to be sorted. [3] It is especially useful and efficient where the number of elements to be sorted is a bounded number. It sorts a string of integer inputs on the basis of their key value. The key value can be any digit of the number, the unit's place, the ten's place et al. It uses an input array of size n, a count array of size k and an output array of size n, where n is the number of inputs and k the number of different key values possible, which is in most cases 0 to 9, a total of 10. [4] Counting sort is an effective way to sorting integers. It can be used as a subroutine with other algorithms such as Radix sort that are capable of handling larger data sets in a better manner. On its own, it is a very powerful algorithm for GPUs [5] and can be modified for parallel computing. The following is a table of complexities of the most commonly used sorting algorithms [6]

Table 1. Complexities of Popular Algorithms

Algorithm	Time Complexity	Space Used
Bubble sort	$O(n^2)$	One array
Selection sort	$O(n^2)$	One array
Insertion sort	$O(n^2)$	One array
Quick sort	O(nlogn)	One array
Counting sort	O(n+k)	Three arrays

The table shows the complexities of various popular sorting algorithms. We can see that the time complexity of counting sort is linear and the sort is efficient. When n is approximately equivalent to k that is the input set is not very large and is almost equal to the number of different possible key values, the time complexity becomes O(n). But this sorting algorithm uses a considerable amount of space. It uses three arrays whereas mostly, others use one. Apart from Counting Sort, Quicksort is a very good option since it works the best among the others, in the average case. Many combinations such as Quicksort combined with insertion sort have shown to give faster results. [7] Many attempts have been made and are continued to be made to reduce the complexity of these sorting algorithms including counting sort thereby making them more efficient. Significant attempts in improving the time complexity have been made, especially for integer sorting algorithms of which Counting Sort is an important example. Paper [8] shows an integer sorting algorithm that sorts a sequence of integers in O(1) time, on a reconfigurable mesh of size n*n. An attempt has been made through this paper to reduce the space complexity of Counting Sort as explained in detail in further sections.

2. LITERATURE REVIEW

Counting Sort is a very efficient sporting algorithm with linear running time, especially efficient with a small input data set. Many attempts have been made to make it more efficient. Counting sort proves to be an efficient algorithm for GPU implementation and much advancement has been made in this field. A modification of integer sort algorithm does not require synchronisation in the last step, hence provides superior performance. [9] A variant of Counting Sort, occurrence sort is especially beneficial as it removes duplicate values and speeds up the computation. Implementations have been performed in Obsidian, a programming language for GPU Programming. Results of paper [5] have many improvements over Library Thrust where Library Thrust is a C++ Library and has similar goals as Obsidian. The process in Obsidian includes creating a histogram and then performing reconstruction. The results implementing counting sort are considerably faster than Library Thrust implementation except in some cases, reason being the fact that number of threads in reconstruction step is proportional to the range. Resultantly, for smaller ranges the execution becomes more and more sequential. The Occurrence Sort implementation is twice as fast as Library Thrust and sometimes even four times as fast, a clear optimization. Comparing the two sorting algorithms, that is occurrence and counting sort in Obsidian implementation, occurrence sort was mostly twice as fast as counting sort. Hence there lies a huge scope for optimizing counting sort. Counting sort is a good and competitive algorithm for sorting keys. It relies on indexing and not comparison as already stated and uses the intermediate count array for pointing a particular input number to its final position in the output array. Hence here as well, it uses three arrays for sorting. Paper [10] shows, integer sort in the form of counting sort implemented on four processors. In the experiments, exponential distribution shows that the sort had to run for at least three seconds before coming to a stop. In these experiments, memory size is important and is seen that it can affect the runtime distributions of jobs. The integer sort on elvis, completed 30% sooner than on caesar. An important notable fact determined was that the amount of memory had more influence on the runtime rather than the speed of the processor. Hence, here too, it shows that the memory occupied by an algorithm is of high importance towards efficiency. Introduction of parallelism proves to be an optimizer of algorithm performance, as shown in [11], in which a parallel RAM model is used that allows concurrent reads and writes. In the integer sort used here, the integer keys are restricted to at most polynomial magnitude. The algorithm costs logarithmic time and the product of the time and processor bounds are bounded by a linear function of the size of the input, unlike previous algorithms that required at least a linear number of processors to achieve a logarithmic time bound. Paper [12] is an attempt to reduce the complexity of counting sort wherein their proposed method, reduces complexity of counting sort to O (n). In this method it uses three arrays as usual, that is the input array [1....n], output array [1....n] and the temporary array instead of the count array that stored the intermediate calculations. It reduces the complexity by not making comparisons. Paper [13] explains counting sort as one of the most efficient algorithms known today. It divides the sequential algorithm into two steps that is namely Distribution and Output. The Distribution step places each element of the input array into its respective bucket or place i the count array. The Output step involves traversing the count array in increasing order of key value for placing elements in increasing order in the output array which is subsequently displayed. Hence here the original counting sort using three arrays is described. Here they have implemented a parallel algorithm for integer sorting for multi-core processors. A variant of Counting Sort namely P-Counting Sort has been used. It distributed code blocks over processors and algorithms such as LoopBSort distribute loop iterations over processors. The time complexity for a parallel sorting algorithm is given by T (N, Kmax, p) = O (N/p + p.Kmax). Here N is the size of the input array, p is the number of processors and Kmax is the key range. P-Counting Sort enables sorting on multi-core processors by using domain decomposition of input data, this was it gives a good parallel sort time. Other variants such as padding sort give an even better parallel sorting time. The results in this paper show that the speed up of Padding sort is 1.4 to 4 times more that the speedup of P-Counting Sort. Apart from the above stated, a hybrid counting sort is an efficient way for accelerating a particle-in-cell. [14] Counting sort proves to be a very efficient algorithm for usage in many fields hence, like other algorithms, it is important to optimize it that is by reducing the memory it consumes and for faster execution.There are many attempts at reducing time complexity and optimizing Counting Sort as it is a very efficient sorting algorithm ad its variants can be applied to various fields and uses. In this paper, a space optimization technique has been introduced, such that instead of three arrays only two have been used, removing the need to use the output array.

3. COUNTING SORT

Counting Sort is a non-comparative, stable integer sorting algorithm that sorts the input numbers on the basis of their keys, that is any one particular digit say one's, ten's et al and it maintains the order of input in the case where two numbers have the same key value. It uses the following the arrays.

Table 2. Arrays used in Counting Sort

Array	Size
Input	N
Count	K
Output	N

Here N is the number of elements in the input array and K is the number of key values possible, ideally 10 that is 0 to 9.

Counting Sort works in the following way:

- Inputs the numbers
- Extracts the desired key value from each input number
- Counts the frequency of occurrence of each key i in the ith position of the count array(intermediate array)
- Calculates the exact position of the key (in turn the number itself in the output array)
- Displays the hence sorted output array

The basic mechanism of counting sort is to count the frequency of occurrences of each key and then performing a prefix sum. It has three steps, constructing a histogram of key frequencies, calculating the start position of each index and then displaying the sorted array. [4][15]

3.1. Pseudocode

The example I will be taking for demonstrating the sorting algorithm is an array of first 500 prime numbers, so since they are already sorted in ascending order , the following code is for un-sorting them, where in[] is the input array:

```
A[500],n←500, i←0,k←0,j←n-1
do
{if(i%2==0)
        A[k]←in[i]
       k++
else
     A[j]←in[i]
     j--
i++
} while (i<n && j>=n/2)
```

The following is the sorting code (according to increasing order of unit's place), in [] is the input array, count[] the intermediate count array and out[] the output array:

int key (int x)//function for extracting key value from each input number, here the unit's place

```
    return (x%10)
for i ←0→ n
    x←in[i]
    count [key(x)] ←count[key(x)]+1
for i←0→k-1
    oc ←count [i]
    count [i] ← nc
    nc←nc+ oc
for i←0→n
    x←in[i]
    out[count[key(x)]] ←x
    count[key(x)]← count[key(x)]+1
```

3.2.Example

For instance let the input array be first 500 prime numbers:

1	2	3	....	3571	3581

After un-sorting the already sorted array, such that it is ready for application of sorting algorithm, the array looks like:

1	3	7	....	5	2

And we want to sort the array in increasing order according to their unit's digit hence the key will be the unit's digit

The count array will look like:

0	125	1	123	0	1	0	127	0	125

After all calculations according to the algorithm, finally, the output array sorted in increasing order of the elements' unit's place will be:

1	11	31	....	3539	3559

3.3.Complexity

The counting sort algorithm has a time complexity of O (n + k). The initialisation of the input and output array of size n takes n loops and count array takes a maximum of k+1, hence complexity becomes O (n + k). If n and k are comparable then the sorting becomes highly efficient and the complexity becomes O (n).The space it requires are three arrays, two of size n and one of size k.

4. PROPOSED SPACE OPTIMIZATION

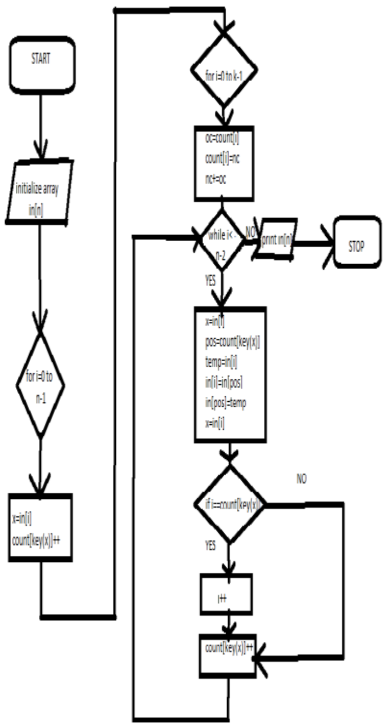
The optimization used here is in terms of space. Instead of using three arrays, that is the input, count and output array, only two arrays have been used.

Table 3. Arrays used in Proposed Space Optimized Counting Sort

Array	Size
Input	N
Count	K

Here Input is used as an input array as well as the final array used for display, swapping is done in such a way as to arrange the numbers in the desired order in the input array itself. Size N is the number of elements of input as usual. The count array is used as it is for intermediate calculations.

4.1.Flow Chart



4.2.Pseudocode

Again, first the code for un-sorting the array of prime numbers:

```
A[500],i←0,k←0,j←499
do
{ if(i%2==0)
        A[k]←in[i]
       k++
else
     A[j]←in[i]
```

```
    j--
i++
} while(i<499 && j>=250)
```

The following is the proposed code, in [] is the input array and count[] the intermediate array

```
int key (int x)
  return (x%10)
for i ←0 →n
    x ←in[i]
    count [key(x)] ← count [key(x)]+1
for i ←0 → k-1
    oc ←count [i]
    count [i] ← nc
    nc ← nc+ oc//position calculated
 i ←0
while i<=n-2
    x ←in[i]
    pos ←count [key(x)]
    temp ←in[i];
    in [i] ←in[pos]
    in [pos] ←temp
    x ←in[i]
    if i ==count[key(x)]
    i ←i+1
    count [key(x)] ← count [key(x)]+1
```

4.3.Example

We take the same example, input array:

1	2	3	....	3571	3581

After applying the un-sorting code:

1	3	7	....	5	2

And we want to sort the array in increasing order according to their unit’s digit hence the key will be the unit’s digit

The count array will look like:

0	125	1	123	0	1	0	127	0	125

After calculating the positions, the count array is:

0	0	125	126	249	249	250	250	377	377

The space optimized algorithm does not use an output array it uses the input and the count array itself, by performing multiple swaps in the input array taking queue from the count array.

So, according to the algorithm, input array/ final array containing sorted elements:

1	11	31	....	3539	3559

Hence the above is the sorted array (sorted on the basis of their unit’s digit) without using an extra output array.

4.4.Complexity and Mathematical Analysis

As depicted in Table1 of this paper, counting sort has a linear time complexity and is one of the most efficient sorting algorithms today. The proposed space optimized counting sort algorithm reduces the number of arrays used without hampering the time complexity of the algorithm. Time complexity remains the same O (n + k) since no subroutines have been called within the modified code.

Number of iterations for Input array initialisation: n

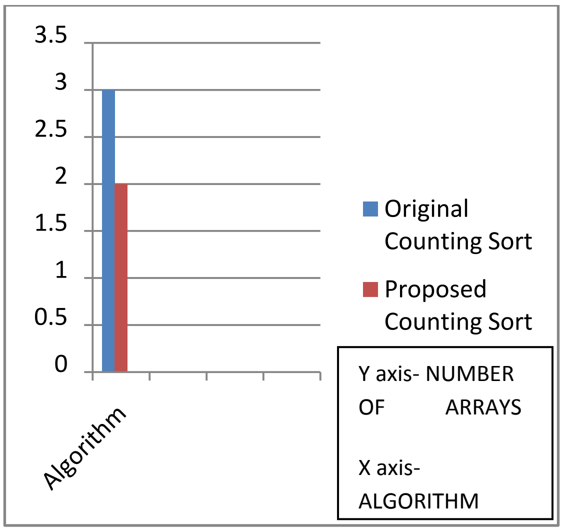
Number of iterations for Count array computation: k+1

The new proposed code runs from 0 to n-2 = n-1 times.

So the total time complexity = O (n + k + 1 +n - 1) = O (n + k)

The space optimization comes by using only two arrays instead of three, thereby reducing the amount of memory required for computation. It uses only size n bits for input and k+1 bits for count array as opposed to extra n bits of memory for the output array. Hence, the total memory/space required by the algorithm is reduced.

Chart 1. Comparison of Space Utilization



By using the proposed algorithm, 33.34% of previously used memory is saved.

Table 4. Comparison of Arrays

Algorithm	Number of Arrays used
Original Counting Sort	2
Proposed Counting Sort	3

Hence, the proposed algorithm proves to be more space efficient.

5. CONCLUSION

Counting Sort is an extremely useful and efficient sorting algorithm. It is used in many diverse fields such GPUs, particle accelerators and many other areas. Due to its importance, numerous attempts have been made to improve its time complexity and the space it uses for execution. The newest optimization is in the energy it takes to execute the algorithms. It has been shown that better performing algorithms consume more energy. Researchers have tried to optimize the algorithm in terms of all the above stated parameters. It is very necessary to optimize the algorithms that we use today. Faster and less space consuming procedures are indispensable for increased efficiency. The algorithm given through this paper uses lesser number of arrays than usual, for Counting Sort thereby reducing the memory required to run the algorithm and at the same time, making it more space efficient. As shown in paper [10], decreased space consumption can also affect run-time; hence the proposed algorithm that uses 33.34% less space than used by the original counting sort can prove to be highly efficient especially in large data sets.

6. REFERENCES

[1] Comparison of Bucket Sort and RADIX Sort, Panu Horsmalahti panu.horsmalahti@tut.fi, June 18, 2012

[2] Energy consumption analysis of parallel sorting algorithms running on multicore systems, Zecena et al

[3] Integer sorting algorithms for coarse-grained parallel machines, AlSabti, K., Ranka, S.

[4] http://video.mit.edu/watch/introduction-to-algorithms-lecture-7-counting-sort-radix-sort-lower-bounds-for-sorting-14155/

[5] Counting and Occurrence Sort for GPUs using an Embedded Language, Josef Svenningsson, Bo Joel Svensson, Mary Sheeran, Dept of Computer Science and Engineering Chalmers University of Technology, {josefs, joels, ms}@chalmers.se

[6] T. H. Cormen, C. E. Leiserson, R.L Rivest, and C. Stein. Introduction to Algorithms". MIT Press, 3nd edition edition

[7] Curtis R. Cook and Do Jin Kim. \Best sorting algorithm for nearly sorted lists". Commun. ACM, 23(11):620-624, November 1980

[8] Integer sorting in O(1) time on an n*n reconfigurable mesh, Olariu, S., Schwing, J.L., Zhang, J., Computers and Communications, 1992. Conference Proceedings, Eleventh Annual International Phoenix Conference

[9] Design and implementation of an efficient integer count sort in CUDA GPUs, Vasileios Kolonias, Artemios G Voyiatzis, George Goulas, Efthymios Housos, Concurrency and Computation: Practice and Experience

[10] The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variances in Run-time Predictions, Robert Armstrong, Hensgen, Debra, Taylor Kidd

[11] An optimal parallel algorithm for integer sorting, Reif, J.H., Foundations of Computer Science, 1985., 26th Annual Symposium

[12] Paper on “Implementation of the technique of Space minimization for Counting Sort algorithm” by Sanjeev Kumar Sharma ,Professor and Dean, JP Institute of Engineering & Technology, Meerut, dean.ar@jpiet.com and Prem Sagar Sharma Research Scholar, Premsagar1987@rediffmaomil.c at Conference on Advances in Communication and Control Systems 2013 (CAC2S 2013)

[13] Parallel Algorithm for Integer Sorting with Multicore Processors, by S. Stoichev and S. Marinova

[14] Accelerating a Particle-In-Cell Simulation using a Hybrid Counting Sort, K. J. Bowers, Electrical Engineering and Computer Science Department University of California at Berkeley

[15] Analysis of Algorithms I: Counting and Radix Sort, Xi Chen, Columbia University